Below is a **refined** “Design and Implementation Documentation” for **Buyzu**, reorganized and partially **condensed** according to your new headings. The content remains **sufficiently detailed** for technical readers but with simpler **category organization**.

**Design and Implementation Documentation**

**for**

**Buyzu**

**Final Version**

Prepared by

Group Name: E7

|  |  |  |
| --- | --- | --- |
| GUO Menglong | 1155191409 | 1155191409@link.cuhk.edu.hk |
| MA Heyang | 1155191433 | 1155191433@link.cuhk.edu.hk |
| XUE Guangxuan | 1155194956 | 1155194956@link.cuhk.edu.hk |
| YAN Yitao | 1155191529 | 1155191529@link.cuhk.edu.hk |
| ZHU Keyu | 1155191834 | 1155191834@link.cuhk.edu.hk |

|  |  |
| --- | --- |
| Instructor: | *Dr.LAM Tak Kei* |
| Course: | CSCI 3100 |
| Date: | 09/05/2025 |

**Table of Contents**

1. Document Introduction 1

1.1 Requirements Overview 1

1.2 Design and Documentation Scope1

1.3 Reference Standards 2

2. System Architecture Design 3

2.1 High-Level Architecture Diagram 3

2.1.1 Frontend Architecture 3

2.1.2 Backend Services 3

2.1.3 Data Storage Layer 5

2.1.4 External Integrations 5

2.2 Key Design Decisions 5

2.2.1 Why Spring Boot 5

2.2.2 Why Vue framework 6

2.2.3 Why MySQL database 6

2.2.4 Security Architecture 6

2.3 Architecture Diagram 7

3. Component Design 8

3.1 User Authentication Module 8

3.2 Product Recommendation Engine 10

3.3 Search Ranking Algorithm 12

3.4 Shopping Cart Service 13

3.5 Checkout Service 15

3.6 Admin Service [New] 16

4. Interface Specifications 20

4.1 API Overview 20

4.2 API Endpoints 20

4.2.1 User Authentication & Management 20

4.2.2 Admin Service [New] 21

4.2.3 Product Discovery & Search 23

4.2.4 Shopping Cart Operations 23

4.2.5 Order Processing 24

4.2.6 Payment Handling 24

4.2.7 Review & Ratings 25

4.2.8 External Integrations 25

4.3 Data Formats 25

4.4 Authentication & Security 26

4.5 Rate Limiting 26

4.6 Error Handling 26

4.7 WebSocket Support 27

4.8 API Documentation & Testing 27

5. Data Model Design 28

5.1 High-Level Database Schema 28

5.1.1 User Table 28

5.1.2 Admin Table [New] 28

5.1.3 Product Table 29

5.1.4 Cart Table 29

5.1.5 Order Table 30

5.1.6 Payment Table 30

5.1.7 Review Table31

5.1.8 Category Table 31

5.2 DATA FLOW 32

5.2.1 User Registration 32

5.2.2 User Login 32

5.2.3 Admin Service [New] 33

5.2.4 Product Browsing and Search 33

5.2.5 Cart Management 33

5.2.6 Checkout and Payment 34

5.2.7 Order Tracking 34

5.2.8 Product Reviews 34

6. User Interface Design 36

6.1 Layout and Flow (“Site Map”) 36

6.2 Accessibility Compliance 37

7. Assumptions 39

7.1 User Base and Traffic Assumptions 39

7.2 Payment and Transaction Processing 39

7.3 Client-Side Compatibility 39

7.4 Security and Compliance 40

7.5 Cross-regional compliance assumption 40

8. Appendices 41

8.1 UML Class Diagram (Main Components) 42

8.2 Version History 44

9. Conclusion45

**1. Document Introduction**

This **Design and Implementation Documentation** is meant to guide **technical developers** in implementing an online shopping platform called “Buyzu.” While it references and satisfies all requirements detailed in the **Software Requirements Specification (SRS)**, the design details within **may evolve** as the software matures. Any **low-level or algorithmically intricate details** not critically tied to top-level design decisions can be found in **code comments**, **API references**, or **implementation notes** within the source repository.

* 1. **Requirements Overview**

The **Buyzu** platform’s SRS specifies:

* **User Management:** Signup, login, secure account handling, password resets, SSO.
* **Product Discovery:** Browsing, filtering, and recommended products.
* **Cart Management and Checkout:** Adding/removing items, adjusting quantities, secure payment.
* **Order Tracking:** Real-time integrations with logistics APIs.
* **Reviews and Ratings:** Post-purchase feedback.
* **Admin Management [New]:** Manage user information, product information and follow up orders

These requirements target both end-users (*e.g.*, customers with minimal technical background) and system stakeholders (developers, project managers, testers). The present document focuses on **how** to implement these functionalities, referencing:

* Internal architectures
* Data flows
* Component responsibilities and their interactions.

**1.2 Design and Documentation Scope**

This design document:

* **Summarizes** top-level system architecture.
* Explains **major component responsibilities** and **internal interfaces**.
* Shows how **external interfaces** (e.g., payment gateways, logistics APIs) connect to
* Uses a **tiered approach**:
  + Start from high-level overviews (e.g., component diagrams).
  + Drill down to detail only where necessary.
* Leaves **fine-grained details** (e.g., exact sorting algorithm if it does not matter) to source-level documentation.

**1.3 Reference Standards**

* OAuth 2.0 Authorization Framework (RFC 6749)
* PCI-DSS v4.0 for payment security
* GDPR Article 32 (Data Encryption Requirements)

**2. System Architecture Design**

**2.1 High-Level Architecture Diagram**

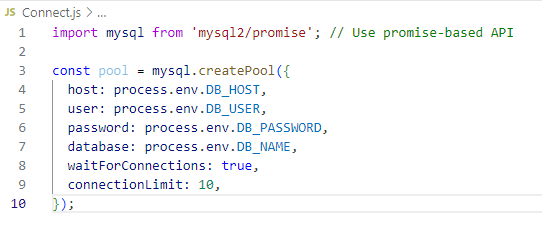
Buyzu adopts a layered microservices architecture to help abstract the functions of systems recursively into levels and ensure scalability & reliability. The system is divided into four core parts:

**1.Front-End (UI/Client)**

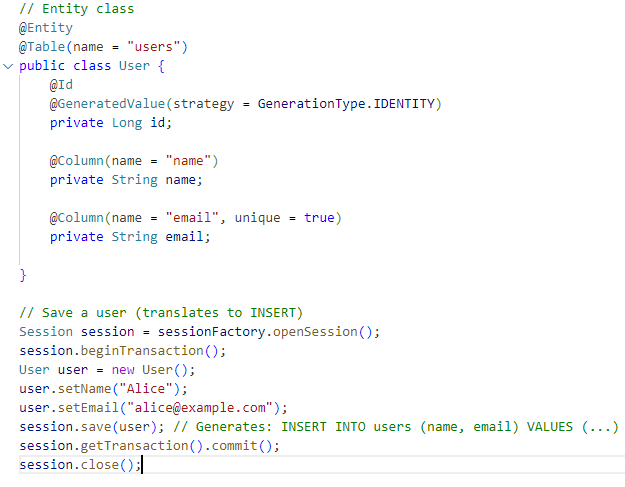
* **Technologies :** JavaScript, HTML, CSS, Vue, Next.js\*
* **Core Features:**
  + A web or mobile interface for users to interact with, developed by HTML and CSS for static layout of interface.
  + Some dynamic UI components like cart synchronization and user sessions by JavaScript.
  + React Single-Page Application (SPA) with Next.js\*
  + Vue framework is applied to improve higher development efficiency.
* **Communication:**
  + Vue.js components make GET/POST requests to Spring Boot controllers via REST APIs (JSON payloads).

**2.Backend (Server-side)**

* **Technologies:** Python,Java, Spring Boot, Node.js, Git, RESTful APIs, Hibernate
* **Core Features:**
  + - Auth service for users sign up/login email/password or social accounts (like Google/WeChat) by Spring Boot.
    - Payment services like Alipay/WeChat Pay integration by Spring Boot.
    - Provides RESTful APIs for user management, product info, cart/checkout, orders, and reviews.
    - We utlize Git throughout the developing process as a version control system to help maintain and track changes in code changes.
* **Communication:**
  + **Data-layer Communication**:
    - We use Node.js to connect to and interact with MySQL database
      * Code example:



* + - Hibernate ORM is applied to automatically map Java objects to SQL queries in MySQL database
      * Code example:



* + - Spring Boot services execute business logic and manage transactions.
  + **Asynchronous Inter-service Communication**
    - **Use Case**: Order Status Updates
    - **Flow**:
      * Order Service publishes an event: order\_shipped (order\_id=666).
      * Notification Service subscribes to the event → Sends SMS/email to the user.
      * Cart Service subscribes → Removes shipped items from the user’s cart.
    - **User Impact**: Real-time notifications without page refresh.
  + **Synchronous Inter-service Communication** 
    - **Use Case**: Payment Verification
    - **Flow**:
      * Payment Service > Auth Service: Validate user’s JWT token.
      * Payment Service > Product Service: Check product availability.
      * Payment Service > Order Service: Create order record on success.
    - **User Impact**: Instant feedback during checkout (e.g., *"*Payment Successful!").

**3. Data Layer**

* **Technologies :** Alipay/WeChat Pay APIs (mTLS + TLS 1.3), credit card processors.
* **Core Features:**
  + Stores structured data for users, products, carts, orders, etc.
  + MySQL 8.0 for transactional data (ACID compliance)
  + Redis 7.0 for session caching and real-time recommendations.
* **Communication:** Accessed via Hibernate/JDBC; ACID-compliant transactions

**4. External Integrations:**

* **Payment Gateways:** Alipay/WeChat Pay APIs (mTLS + TLS 1.3), credit card processors.
* **Logistics APIs:** SF Express/ZTO for shipping status updates
* **Security:** HSM-managed keys, rate limiting.
* **Analytics:** Possibly future integration for recommendations or usage tracking.

**2.2 Key Design Decisions**

1. **Why Spring Boot:**
   * Simplified security configuration via Spring Security
   * Native support for OAuth 2.0 and JWT
   * Kubernetes compatibility for horizontal scaling.
   * Help to save some time, which may largely improve efficiency and reduce the development costs.
2. **Why Vue framework:** 
   * Enables reusable, modular components (e.g., cart, sessions) with reactive data binding, streamlining dynamic feature implementation and maintenance.
   * Boosts development efficiency through declarative syntax and streamlined state management.
3. **Why MySQL database:**
   * Stable service and higher performance
   * Lightweight, easy to install and use, lower maintenance costs.
   * Supports multiple operating systems, provides various API interfaces, and offers excellent support for Java (Our main language in back end).
4. **Security Architecture:**
   * End-to-end Encryption: TLS 1.3 for all external/internal communications.
   * Key Management: HSM-stored JWT secrets and payment credentials.
   * Rate Limiting:
     + - * Public APIs: 1,000 requests/minute.
         * Authenticated APIs: 5,000 requests/minute.

**2.3 Architecture Diagram**

A simplified block diagram:

┌──────── Front-End ────────┐

│ Client (Browser) │

│ or Mobile Application │

└───────────▲───────────────┘

│ HTTP/HTTPS

▼

┌───────── Backend (Express) ─────────┐

│ - User Controllers │

│ - Product Controllers │

│ - Cart & Checkout Controllers │

│ - Order Controllers │

│ - Review Controllers │

└─────────────────────────────────────┘

│

▼

(Business Logic)

│

▼

┌─Data Layer──┐ ┌── External Integrations ──┐

│ MySQL DB │ │ Payment Gateways │

│ Relational)│ │ / Logistic APIs │

└─────────────┘ └───────────────────────────┘

**3. Component Design**

**3.1 User Authentication Module**

1. **Responsibilities**
   * Implement a user registration/login process that supports multi-method authentication (email-password, OAuth 2.0, etc.) [modified].
   * Ensure secure storage and encryption of user login credentials to maintain user identity legitimacy.
2. **Input Specifications [modified]:**
   * r***egistration\_method*: An enum type, could be chosen within email, phone or OAuth**
   * ***username*: A string type that uniquely identifies the user, bi-directionally bound to *user\_id***
   * ***password*:** **A string** type**, which must pass the password strength check (length ≥ 8, including case + numbers)**
   * *email (if* r***egistration\_method=email)*: A string** type indicates user’s email address**, which must be verified later through MFA**
   * *email\_code (if* r***egistration\_method=email)*: An integer(int32) type, inputted by user according to the** captcha sent to the assigned email address
   * *OAuth\_provider (if* r***egistration\_method=OAuth)*: An enum type which determined by user from supported third-party providers (e.g. Google, Facebook, etc.)**
   * *access\_token (if* r***egistration\_method=OAuth)*: A string type returned after the identity service provider successfully completing the authentication progress**
   * *~~captcha~~***~~: A string type reads from user’s response to the complicated graphic captcha demonstrated~~ Adoption of email verification link instead [modified]**
   * *terms***: A boolean type suggests whether user accept the terms of service**
3. **Output Specifications:**
   * *registration/login****\_status*: An enum type, could vary within success, pending or error**
   * *user\_id*: A string type which stands for user’s unique identity regardless the registration method
   * *user\_information*: An object type stores user’s account information after encryption
   * *session\_token:* A string type valid for short-term access
   * *error\_code***: A string type explains the reason of failure**
   * *OAuth\_profile***: An object type returned by the third-party providers, includes user’s data**
4. **Algorithms:**

The main algorithms used in this section are the hash algorithm ~~(Argon2)~~ for encryption and the JWT token generation and authentication for secure sessions. In addition to this, the user may face some multi-factor authentication. The main registration process is that the user first selects the registration method, then enters a series of information and the algorithm verifies it, then gives the user a unique id and generates a user object and finally stores the encrypted user data. The principle of login is similar, users need to choose the login method, password and MFA authentication, and generate an access token after successful authentication.

1. **Classes (Core) [modified]:**

@dataclass

class UserInformation:

encrypted\_data: str

@dataclass

class OAuthProfile:

profile\_data: Dict[str, Any]

@dataclass

class User:

user\_id: int

username: str

password\_hash: str

email: Optional[str] = None

**3.2 Product Recommendation Engine**

1. **Responsibilities:**

**The Product Recommendation Engine is a critical component within the e-commerce platform Buyzu, responsible for:**

* + **Personalized Recommendations based on Latest Trends: Buyzu's product recommendation engine analyzes the user's browsing history, behavioral patterns, and purchasing records to customize the recommendation of popular products that align with current tastes and trends, giving the user an ideal shopping experience and a better understanding of celebrity style.**
  + **Adaptive Recommendation Algorithms for Users with Different Frequency: For first-time users or less frequent purchasers, meta-learning is used to quickly establish user labels; For frequent users, the recommendation experience is optimized using algorithms based on mixed language processing and graphical models.**
  + **~~Cross-Paradigm Fusion~~**~~: Integrating structured (e.g., user profiles) and unstructured data (e.g., social media trends) for robust modeling.~~ Only focus on user history now (e.g., purchasing history, shopping cart, browse history) [modified]
  + **Performance Optimization**: Balancing accuracy, diversity, and computational efficiency at scale.

1. **Input Specifications [modified]:**
   * ***user\_id*: An integer(int32) type that uniquely identifies the user** **as an index value of *username***
   * ***recent\_interactions*:** **An array of integers (int32) storing the user's 50 most recent interactions**
   * *context***: Nested *UserContext* type, contains context information such as device, location, time, etc.**
2. **Output Specifications [modified]:**
   * *items***: An array of *RankedItem* type, stores** the ordered list of recommendations generated by the system, typically sorted by relevance or predicted preference
   * ***metadata*:** Additional details about the recommendation engine, such as the model version used to generate these recommendations, and potential performance metrics
3. **Algorithms [modified]:**

This algorithm borrows from existing recommendation algorithms and mainly uses quantitative data abstracted from users' past behaviors. The algorithm first uses the NGCF algorithm to extract the deep relationships in the user's historical behaviors, especially focusing on the timing of the behaviors, to form a more accurate profile of the user. After that, the BERT-based language model analyzes the potential connections between the items of interest to the user. The function scoring mechanism balances the scores given by the two algorithms to give a list of recommended items that better match the user's interests.

1. **Classes (Core) [modified]:**

from sklearn.neighbors import NearestNeighbors

def get\_baseline\_candidates(vec,meta,hist,id2idx):

nbrs=NearestNeighbors(metric="cosine",algorithm="brute")

nbrs.fit(vec)

idx2pid=meta["productID"].tolist()

cands\_dict={}

for uid,seq in hist.items():

user\_vec=normalize(vec[[id2idx[i] for i in seq]].mean(0,keepdims=True))

\_,ind=nbrs.kneighbors(user\_vec,n\_neighbors=min(KNN,len(vec)))

cands=[int(i) for i in ind[0] if idx2pid[i] not in seq][:TOPK]

cands\_dict[uid]=cands

return cands\_dict

class SASRec(nn.Module):

def \_\_init\_\_(self,emb):

super().\_\_init\_\_()

n,dim=emb.shape

self.static=nn.Embedding.from\_pretrained(torch.tensor(emb),freeze=True)

self.proj=nn.Linear(dim,HIDDEN,bias=False)

self.pos=nn.Embedding(MAXLEN,HIDDEN)

layer=nn.TransformerEncoderLayer(HIDDEN,N\_HEAD,4\*HIDDEN,

batch\_first=True)

self.enc=nn.TransformerEncoder(layer,N\_BLOCK)

self.norm=nn.LayerNorm(HIDDEN)

def forward(self,seq,seqlen):

x=self.proj(self.static(seq))+self.pos(

torch.arange(seq.size(1),device=seq.device))

x=self.norm(x)

x=self.enc(x)

o=x[torch.arange(x.size(0)),seqlen]

logits=o @ self.proj(self.static.weight).T

return logits

**3.3 Search Ranking Algorithm**

1. **Responsibilities:**

It is responsible for semantically cutting the user input, processing the links between the different parts, calculating the relevance score between the query and the candidate items using synonym association maximization and feeding it back to the user in descending order, as well as correcting possible spelling errors.

1. **Input Specifications:**

* ***query*: A string type that describes user’s intention**
* ***filters*:** **An object type containing user’s filtering criteria for the search**
* *sort\_by***: A string defining how the results should be sorted, could be chosen actively by user**

1. **Output Specifications:**

* ***results*: An array of product objects, containing the following attributes:**
* ***product\_id*:** A string type uniquely identifying the product
* ***title***: A string type returning the product’s name
* ***relevance***: A float type indicating how robustly the product connects to user’s query
* ***price***: A float type displaying the product’s price
* ***match****\_option*: An object type showing whether the product is matched by title or description

1. **Algorithms:**

Firstly, the BM25 algorithm is used to calculate the relevance of the query content and the text in the product library, weighted to take into account the price and classification, and then display a mixture of title-matched and description-matched product information in decreasing order.

1. **Classes (Core):**

class SearchService:

def \_\_init\_\_(self, repo: ProductRepository, ranker: RankingEngine):

self.repo = repo

self.ranker = ranker

def execute\_search(self, query: SearchQuery) -> SearchResult:

analyzed\_query = QueryAnalyzer.parse(query.raw\_query)

products = self.repo.find\_products(analyzed\_query)

ranked\_products = self.ranker.rank(products, analyzed\_query)

return paginate(ranked\_products, query.page, query.page\_size)

class BM25Ranker:

def \_\_init\_\_(self, k1=1.5, b=0.75):

self.k1 = k1

self.b = b

def build\_index(self, corpus: List[str]):

pass

def score(self, document: str, query\_terms: List[str]) -> float:

pass

**3.4 Shopping Cart Service**

1. **Responsibilities:**

Allows the user to add items of interest to a list called shopping cart during browsing and later modify, sort, delete, etc., and allows the user to check out all or some of the items in the shopping cart.

1. **Input Specifications:**

* ***user\_id*: An integer(int32) type that uniquely identifies the user** **as an index value of *username***
* ***product\_id*:** A string type uniquely identifying the product
* ***initial\_quantity:* An integer(int32) type representing the quantity acquired by user**
* ***new\_quantity*: An integer(int32) type representing the updated quantity acquired by user**
* ***cart\_item\_id*: A string type that uniquely specifies a valid shopping cart item**
* *sort\_by***: A string defining how the results should be sorted, could be chosen actively by user**
* *selected\_items***: A string array storing the aimed items and quantities to be forwarded to checkout procedure by user**

1. **Output Specifications:**

* ***current\_cart\_items*: An array of product objects, containing the current *cart\_item*sand their distinctive *id*s**
* ***invalid\_cart\_items*: An array of product objects, containing the *cart\_item*sthat are no longer available**
* ***product\_id*:** A string type uniquely identifying the product
* ***total\_price***: A float type returning the total price of the *selected\_items*
* ***checkout\_status***: An enum type indicating whether the *selected\_items* are available to proceed to the checkout step
* *error\_code***: A string type explains the reason of failure**

1. **Algorithms:**

When adding a new item to the shopping cart, the order of time stamps should be followed to determine the order in which the items are displayed. Each time a new item is added, first the inventory should be checked against the id of the item, if the quantity entered by the user is not greater than the inventory quantity, then add is performed, otherwise an error is returned, and the user is told to modify the quantity. When modifying an item, determine if there is enough inventory according to the new quantity entered by the user. When deleting an item, it will delete the selected item according to the id of the item selected by the user. At the same time, the shopping cart will automatically check if the items currently added to the cart are still valid at short intervals, and the invalid items will be moved to ***invalid\_cart\_items***.

1. **Classes (Core):**

def sort\_cart(items: List[CartItem], strategy: SortKey) -> List[CartItem]:

if strategy == AI\_RANK:

ranked = MLModel.predict([item.features for item in items])

return [item for \_, item in sorted(zip(ranked, items), reverse=True)]

else:

return sorted(items, key=\_get\_sort\_key(strategy))

def resolve\_cart\_conflict(user\_cart: Cart, stored\_cart: Cart) -> Cart:

merged = deepcopy(stored\_cart)

for item in user\_cart.items:

if item.deleted:

merged.remove(item)

else:

existing = next((x for x in merged if x.item\_id == item.item\_id), None)

if existing and item.timestamp > existing.timestamp:

existing.quantity = item.quantity

return merged

**3.5 Checkout Service**

1. **Responsibilities:**

It is responsible for finalizing the price of the product information passed by the user from the shopping cart and completing the payment of the order through the payment portal, and tracking the logistics information after the payment is successful.

1. **Input Specifications:**

* ***user\_id*: An integer(int32) type that uniquely identifies the user** **as an index value of *username***
* *selected\_items***: A string array storing the aimed items and quantities to be forwarded to checkout procedure by user**
* ***payment\_method:* An enum type suggests the preferred payment method indicates by user**
* ***shipping\_address*: Nested *address* type, including user’s delivery address and geolocation**
* ***discount*: A string type that accept user’s inputted promotion codes**
* *gift\_card***: Nested *giftcard* type that accept valid, non-negative balance gift cards owned by user**

1. **Output Specifications:**

* ***order\_id*: A string type of immutable order identifier, could be used to track the order stauts**
* ***order\_status*: An enum type** indicating whether the order is successfully established
* ***receipt*:** An array of string type which uniquely identifying the purchase action, including the transaction id, the amount charged, and the items bought etc.
* ***shipping\_info*: A nested type contains the tracking information provides by third-party deliver carriers**
* *error\_code***: A string type explains the reason of failure**

1. **Algorithms:**

The checkout service first verifies item availability, pricing consistency, and cart integrity, and then uses optimistic locking to place a temporary hold on inventory to prevent overselling. Once this step is complete, the payment is authorized through a third-party gateway and the authorization token is stored. Upon successful capture, the payment authorization is converted to a final charge, the reserved inventory is converted to a permanent deduction, and the order record is written to the database. The order record is also sent downstream for synchronization with logistics providers for shipment and other subsequent operations.

1. **Classes (Core):**

def process\_checkout(order\_data):

try:

validate\_cart(order\_data.cart)

reserve\_inventory(order\_data.cart)

hold\_payment(order\_data.payment)

confirm\_payment()

deduct\_inventory()

create\_order\_record()

send\_confirmation\_email()

update\_analytics()

except CheckoutError as e:

compensate\_payment\_hold()

restore\_inventory()

log\_failed\_attempt()

**3.6 Admin Service [New]**

1. **Responsibilities:**

The Admin Service first verifies that the caller is a legitimate administrator and checks his/her permissions; then allows the administrator to create, read, update and delete user accounts; add products, modify existing product details, remove products, and query product lists; view details of any orders, change order status, or cancel orders if required; and for each action taken, automatically logs the person who performed it, when it occurred, and what was changed to allow audit and traceability of all administrative actions. For each action taken, an automatic record of who performed it, when it occurred, and what was changed is made available for auditing and tracing all administrative actions.

1. **Input Specifications:**

* ***admin\_id*: An integer(int32) type that uniquely identifies the user** **as an index value of *username***
* *action\_type***: A enum identifying the operation kind, e.g. CREATE\_PRODUCT**
* ***target\_id:* A string type suggests the** ID of the user/product/order to operate on
* ***payload\_object*:** Nested data required for the chosen action

1. **Output Specifications:**

* ***operation\_status*: An enum type** indicating whether the order is successfully established
* ***receipt*:** An array of string type which uniquely identifying the purchase action, including the transaction id, the amount charged, and the items bought etc.
* ***result\_entity*: An object contains** the created or updated user/product/order record, null on delete or failure
* *error\_code***: A string type explains the reason of failure**

1. **Algorithms:**

In the algorithm, the managed service first validates the administrator's credentials and checks their authorization for the requested action. Once authorized, it validates the input payload against the expected patterns and business rules. It then creates an audit log of the “pending” status, recording who took the action, when the action was taken, and the purpose of the action. Next, it assigns the request to the appropriate processor - user management, product management, or order management - to perform the actual CRUD or status update operation on the database transaction. If each step completes successfully, the service commits the transaction, updates the audit log to “Success”, and sends back the newly created or modified entity. If any validation or database operation fails, it backs out of the transaction, marks the audit log as “Failed” with an error code and message, and sends back the standard error response.

1. **Classes (Core):**

class AdminService:

def process\_request(self, admin\_request):

try:

self.authenticate\_admin(admin\_request.admin\_id)

self.authorize\_action(admin\_request.admin\_id, admin\_request.action\_type)

self.validate\_payload(admin\_request.action\_type, admin\_request.payload)

audit\_id = self.start\_audit\_log(admin\_request)

# dispatch

if admin\_request.action\_type in ('CREATE\_USER','UPDATE\_USER','DELETE\_USER'):

result = self.manage\_user(admin\_request)

elif admin\_request.action\_type in ('CREATE\_PRODUCT','UPDATE\_PRODUCT','DELETE\_PRODUCT'):

result = self.manage\_product(admin\_request)

else:

result = self.manage\_order(admin\_request)

self.complete\_audit\_log(audit\_id, success=True)

return { 'success': True, 'result\_entity': result, 'error\_code': None, 'error\_message': None }

except AdminError as e:

self.complete\_audit\_log(audit\_id, success=False, code=e.code, message=str(e))

return { 'success': False, 'result\_entity': None, 'error\_code': e.code, 'error\_message': e.message }

def authenticate\_admin(self, admin\_id):

# look up admin, verify session/token

pass

def authorize\_action(self, admin\_id, action\_type):

# RBAC check

pass

def validate\_payload(self, action\_type, payload):

# schema & business rules

pass

def start\_audit\_log(self, req):

# insert log entry with state = "pending"

pass

def complete\_audit\_log(self, audit\_id, success, code=None, message=None):

# update log entry

pass

def manage\_user(self, req):

# create/update/delete user record in UserDB

pass

def manage\_product(self, req):

# create/update/delete product record in ProductDB

pass

def manage\_order(self, req):

# view or update order in OrderDB

pass

**4. Interface Specifications**

This section defines the interface specifications of Buyzu, covering API endpoints, request/response formats, and interaction mechanisms between different system components.

### **4.1 API Overview**

Buyzu follows a **RESTful API** approach with JSON-based data exchange. The backend provides various microservices, each responsible for specific functionalities. The core API categories include:

1. **User Authentication & Management**
2. **Admin Service [New]**
3. **Product Discovery & Search**
4. **Shopping Cart Operations**
5. **Order Processing**
6. **Payment Handling**
7. **Review & Ratings**
8. **External Integrations (e.g., Logistics, Payment Gateways)**

All endpoints use **HTTPS** for security, and authentication is handled by **JWT tokens**.

The following API endpoint will ignore our website domain name for simplicity. For example, <https://buyzu.com/api/products> will be shown as /api/products

### **4.2 API Endpoints**

#### **4.2.1 User Authentication & Management**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Description** | **Request Body** | **Response** |
| /api/auth/register | POST | Registers a new user | { "username": "user1", "email": "[user1@example.com](mailto:user1@example.com)", "password": "hashed\_pass" } | { "user\_id": "12345", "status": "success" } |
| /api/auth/login | POST | Logs in a user and issues JWT token | { "email": "[user1@example.com](mailto:user1@example.com)", "password": "hashed\_pass" }  OR  { " username": " user1”, "password": "hashed\_pass" } | { "token": "jwt\_token", "user\_id": "12345" } |
| /api/auth/logout | POST | Logs out a user | { "token": "jwt\_token" } | { "status": "success" } |
| /api/auth/reset-password | POST | Initiates password reset process | { "email": "[user1@example.com](mailto:user1@example.com)" } | { "status": "email\_sent" } |
| /api/auth/oauth/login | POST | OAuth login via third-party providers | { "provider": "Google", "access\_token": "oauth\_token" } | { "token": "jwt\_token", "user\_id": "12345" } |

#### **4.2.2 Admin Service [New]**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Description** | **Request Body** | **Response** |
| /api/admin/products | GET | List all products | { "name":"Sneakers", "description":"Running shoes", "price":699.00, "stock":50 } | { "product\_id":"p088", "status":"success" } |
| /api/admin/products | POST | Create a new product |  | { "product": { "product\_id":"p001", "name":"T-shirt", "price":199.00, "stock":120, "description":"Cotton T-shirt" } } |
| /api/admin/products/{product\_id} | GET | Get details for one product |  | { "product": { "product\_id":"p001", "name":"T-shirt", "price":199.00, "stock":120, "description":"Cotton T-shirt" } } |
| /api/admin/products/{product\_id} | PUT | Update an existing product | { "price":179.00, "stock":200, "description":"Discounted cotton T-shirt" } | { "status":"success", "product": { "product\_id":"p001", "name":"T-shirt", "price":179.00, "stock":200 } } |
| /api/admin/products/{product\_id} | DELETE | Delete a product |  | { "status":"success" } |
| /api/admin/orders | GET | List all orders |  | { "orders":[ { "order\_id":"o1001", "user\_id":"123", "status":"PAID", "total":499.00 }, … ], "total":156 } |
| /api/admin/orders/{order\_id} | GET | Get details for one order |  | { "order": { "order\_id":"o1001", "user\_id":"123", "items":[…], "status":"PAID", "shipping\_info":{…} } } |
| /api/admin/orders/{order\_id} | PUT | Update order status | { "new\_status":"SHIPPED", "reason\_code":"DISPATCHED" } | { "order\_id":"o1001", "status":"SHIPPED" } |
| /api/admin/orders/{order\_id}/cancel | POST | Cancel an order | { "reason\_code":"OUT\_OF\_STOCK" } | { "order\_id":"o1001", "status":"CANCELLED" } |

#### **4.2.3 Product Discovery & Search**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Description** | **Request Body** | **Response** |
| /api/products | GET | Get all products (paginated) | ?page=1&limit=20&category=electronics | { "products": [...], "total": 200 } |
| /api/products/{id} | GET | Get product details | N/A | { "product\_id": "P123", "name": "Laptop", "price": 999.99 } |
| /api/products/search | GET | Search for products | ?query=laptop&sort=price | { "results": [...], "total": 50 } |
| /api/products/recommendations/{user\_id} | GET | Get personalized recommendations | N/A | { "recommendations": [...], "algorithm": "SRSTec+NGCF" } |

#### **4.2.4 Shopping Cart Operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Description** | **Request Body** | **Response** |
| /api/cart | GET | Fetch user’s cart items | Authorization: Bearer jwt\_token | { "cart": [ { "product\_id": "P123", "quantity": 2 } ] } |
| /api/cart/add | POST | Add item to cart | { "product\_id": "P123", "quantity": 1 } | { "status": "added" } |
| /api/cart/update | PUT | Update cart item quantity | { "product\_id": "P123", "quantity": 3 } | { "status": "updated" } |
| /api/cart/remove | DELETE | Remove item from cart | { "product\_id": "P123" } | { "status": "removed" } |

#### **4.2.5 Order Processing**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Description** | **Request Body** | **Response** |
| /api/orders | POST | Create a new order | { "cart\_id": "C123", "payment\_method": "Alipay" } | { "order\_id": "O123", "status": "processing" } |
| /api/orders/{id} | GET | Fetch order details | N/A | { "order\_id": "O123", "status": "shipped" } |
| /api/orders/{id}/cancel | POST | Cancel an order | { "reason": "Changed my mind" } | { "status": "cancelled" } |

#### **4.2.6 Payment Handling**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Description** | **Request Body** | **Response** |
| /api/payment/initiate | POST | Initiate payment | { "order\_id": "O123", "method": "CreditCard" } | { "payment\_id": "T987", "status": "pending" } |
| /api/payment/status/{id} | GET | Check payment status | N/A | { "payment\_id": "T987", "status": "success" } |

#### **4.2.7 Review & Ratings**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Description** | **Request Body** | **Response** |
| /api/reviews/product/{product\_id} | GET | Get product reviews | N/A | { "reviews": [...] } |
| /api/reviews/add | POST | Submit a review | { "product\_id": "P123", "rating": 5, "comment": "Great product!" } | { "status": "submitted" } |

#### **4.2.8 External Integrations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Description** | **Request Body** | **Response** |
| /api/logistics/track/{order\_id} | GET | Track shipment status | N/A | { "order\_id": "O123", "status": "In Transit", "tracking\_number": "T56789" } |
| /api/logistics/estimate | POST | Get shipping cost estimate | { "destination": "Hong Kong", "weight": 2.5 } | { "cost": 15.99 } |

### **4.3 Data Formats**

#### **General Response Structure**

{

"status": "success",

"data": { },

"error": null

}

For errors:

{

"status": "error",

"error": {

"code": "INVALID\_INPUT",

"message": "Email is required."

}

}

### **4.4 Authentication & Security**

All authenticated routes require a **JWT token** in the Authorization header:

Authorization: Bearer <jwt\_token>

* JWT tokens expire after **24 hours**.
* Refresh tokens are available at /api/auth/refresh.

### **4.5 Rate Limiting**

* **Public APIs**: 1000 requests per minute.
* **Authenticated APIs**: 5000 requests per minute.
* **Abuse Detection**: IP-based and account-based rate limiting.

### **4.6 Error Handling**

|  |  |
| --- | --- |
| **Error Code** | **Description** |
| 400 | Bad Request (invalid input format) |
| 401 | Unauthorized (invalid/missing token) |
| 403 | Forbidden (insufficient permissions) |
| 404 | Not Found (resource does not exist) |
| 500 | Internal Server Error |

### **4.7 WebSocket Support**

For **real-time notifications**, Buyzu provides WebSocket endpoints:

* /ws/notifications: Order updates, payment status changes.

### **4.8 API Documentation & Testing**

Buyzu's API is documented using **Swagger/OpenAPI**:

* URL: <https://buyzu.com/api/docs>
* API testing via **Postman** or cURL.

**5. Data Model Design**

**5.1 High-Level Database Schemas**

The database schema is designed to support the core functionalities outlined in the SRS, including user management, product discovery, cart operations, order processing, and reviews. Below are the primary tables and their relationships:

**5.1.1 User Table**

User table stores user account information.

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Constraints/Description |
| UserID | INTEGER | Primary Key, Auto-increment |
| Username | VARCHAR(50) | Unique, 4-50 characters (letters, numbers, underscores) |
| PasswordHash | BINARY(60) | bcrypt encryption, min 8 chars (uppercase, lowercase, numbers) |
| Email | VARCHAR(255) | RFC 5322 compliant, verified |

* + 1. **Admin Table [New]**

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Constraints/Description |
| Roles | VARCHAR(100) | Comma-separated list of admin roles (e.g. “superadmin,product\_mgr”); at least one required |
| Status | ENUM('ACTIVE','SUSPENDED','DEACTIVATED') | Default 'ACTIVE'; controls whether the admin can log in/act |
| CreatedAt | DATETIME | Default CURRENT\_TIMESTAMP |
| UpdatedAt | DATETIME | Default CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP |
| LastLogin | DATETIME | Nullable; timestamp of last successful login |
| FailedLoginCount | INTEGER | Default 0; increments on each failed login attempt |

* + 1. **Product Table**

Product Table stores product details.

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Constraints/Description |
| ProductID | CHAR(12) | Primary Key, EAN-13 format |
| ProductName | VARCHAR(100) |  |
| Description | TEXT |  |
| Price | DECIMAL(10,2) | ≥0.01 |
| CategoryID | INTEGER | Foreign Key to Category |
| InventoryCount | INTEGER | ≥0, "Out of Stock" if 0 |

* + 1. **Cart Table**

Cart Table manages items added to the shopping cart.

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Constraints/Description |
| CartID | INTEGER | Primary Key, Auto-increment |
| UserID | INTEGER | Foreign Key to User |
| ProductID | CHAR(12) | Foreign Key to Product |
| Quantity | INTEGER | ≥1 |

* + 1. **Order Table**

Order Table tracks order details and status.

|  |  |  |
| --- | --- | --- |
| Filed Name | Data Type | Constraints/Description |
| OrderID | CHAR(36) | Primary Key, UUID format |
| UserID | INTEGER | Foreign Key to User |
| TotalAmount | DECIMAL(10,2) | Sum of product prices + shipping |
| ShippingAddress | TEXT | Validated format (country, postal code) |
| OrderStatus | ENUM | Values: Processing/Shipped/Delivered/Cancelled |
| PaymentTransactionID | VARCHAR(36) | Foreign Key to Payment |

* + 1. **Payment Table**

Payment Table records payment transactions.

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Constraints/Description |
| TransactionID | VARCHAR(36) | Primary Key, UUID format |
| OrderID | CHAR(36) | Foreign Key to Order |
| Amount | DECIMAL(10,2) | Matches Order.TotalAmount |
| PaymentMethod | ENUM | Values: Credit Card/Alipay/WeChat Pay |
| Status | ENUM | Values: Success/Failed/Pending |

* + 1. **Review Table**

Review Table stores product ratings and reviews.

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Constraints/Description |
| ReviewID | INTEGER | Primary Key, Auto-increment |
| UserID | INTEGER | Foreign Key to User |
| ProductID | CHAR(12) | Foreign Key to Product |
| Rating | INTEGER | 1-5 |
| Comment | TEXT | Optional |

* + 1. **Category Table**

Category Table supports product categorization for browsing/filtering.

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Constraints/Description |
| Constraints/Description | INTEGER | Primary Key, Auto-increment |
| CategoryName | VARCHAR(50) | Unique |

Summary of Relationships:

User Table: Linked to Cart, Order, Review, and Session tables via UserID.

Product Table: Linked to Cart and Review tables via ProductID.Linked to Category table via CategoryID.

Order Table: Linked to Payment table via PaymentTransactionID.

Payment Table: Linked to Order table via OrderID.

* 1. **Data Flow**

The data flow aligns with the key user journeys described in the SRS. Below is a step-by-step breakdown:

**5.2.1 User Registration**

Flow:

1. User enters Username, Email, and Password via the registration form.

2. System checks uniqueness of Email in the User table.

3.If unique:

A verification email is sent to the user.

User clicks the verification link → System creates a new entry in the User table.

4. For SSO registration:

User selects a provider (e.g., Google) → Redirected to OAuth 2.0 flow.

Provider returns user data (e.g., email, name) → System creates/updates the User table.

Tables Involved:

User: Stores user credentials.

Session: Generates an authentication token post-verification.

**5.2.2 User Login**

Flow:

1.User enters Username/Password or selects SSO.

2. System verifies credentials against the User table.

Tables Involved:

User: Validates credentials.

**5.2.3 Admin Service [New]**

Flow:

1. Admin submits Username+Password (or SSO/OAuth).  
2. System looks up the Admin table by Username.  
3. If Admin.Status ≠ ACTIVE, reject with “account suspended/deactivated.”  
4. Verify PasswordHash (bcrypt).

5. On success:  
• Reset Admin.FailedLoginCount → 0  
• Update Admin.LastLogin → current timestamp  
• Issue JWT (including AdminID and Roles)  
• Write a “login\_success” entry in AdminAuditLog (AdminID, timestamp, IP, client)  
• Return { token, admin\_id, roles }

Tables Involved:

• Admin  
– lookup/verify credentials, status, MFA settings  
– update LastLogin, FailedLoginCount, Status  
• AdminAuditLog  
– record every login attempt with outcome, timestamp, IP, client info

**5.2.4 Product Browsing and Search**

Flow:

1. User enters keywords or applies filters (e.g., price range).

2. System queries the Product and Category tables to retrieve matching products.

3. Results are displayed with product details (name, price, image).

4. User interactions (e.g., clicks, views) are logged for recommendation algorithms.

Tables Involved:

Product: Stores product metadata.

Category: Enables filtering by category.

**5.2.5 Cart Management**

Flow:

1. User adds a product to the cart → System checks Product.InventoryCount.

2. If inventory is available:

A new entry is created in the Cart table with UserID, ProductID, and Quantity.

Real-time total price is calculated using Cart.Quantity × Product.Price.

3. User modifies quantity or removes items → Cart table is updated.

Tables Involved:

Cart: Tracks cart items.

Product: Provides real-time inventory and pricing.

**5.2.6 Checkout and Payment**

Flow:

1. User initiates checkout → Cart items are converted into an Order entry.

2. System calculates TotalAmount (product prices + shipping fees).

3. User enters ShippingAddress → Validated and stored in the Order table.

4. ayment details are sent to a third-party gateway (e.g., Alipay).

5. On successful payment:

A Payment record is created with TransactionID and Status.

Product.InventoryCount is decremented.

Order.OrderStatus is set to Processing.

Tables Involved:

Order: Tracks order details.

Payment: Records transaction data.

Product: Updates inventory.

**5.2.7 Order Tracking**

Flow:

1. System polls logistics APIs (e.g., SF Express) for real-time updates.

2. Received data (e.g., "Shipped") updates Order.OrderStatus.

3. Users view status changes via their account dashboard.

Tables Involved:

Order: Stores and updates status.

**5.2.8 Product Reviews**

Flow:

1. After order delivery, user submits a rating and review.

2. System validates eligibility (user must have purchased the product).

3. A new entry is added to the Review table with UserID, ProductID, and Rating

4. Reviews are displayed on the product details page.

Tables Involved:

Tables Involved:

Order: Validates purchase history.

Key Design Considerations:

1. ACID Compliance: Transactions (e.g., inventory reduction + order creation) are atomic to prevent partial updates.

2. Index Optimization: Indexes on User.Username, Product.ProductID, and Order.UserID to accelerate queries.

3. Security: Sensitive fields (e.g., PasswordHash, AuthToken) are encrypted using AES-256 and TLS 1.3.

4. Scalability: Database sharding for Order and Product tables to handle high traffic during sales events.

**6. User Interface Design**

**6.1 Layout and Flow (“Site Map”)**

The user interface of Buyzu is designed to provide a seamless and intuitive experience for users. Below is the layout and flow of the system, along with technical implementation details:

1. **Landing Page**
   * **Functionality**: Displays featured products and promotions, with links for login and registration.
   * **Implementation**:
     1. Use HTML5 and CSS3 to create a responsive layout, ensuring compatibility across devices.
     2. Use JavaScript to dynamically load featured product data (fetched via API from the backend).
     3. Login/registration links should use <a> tags to redirect to the respective pages.
2. **Product Listing/Search Page**
   * **Functionality**: Allows users to browse products by category or apply filters (e.g., price range), and supports keyword search.
   * **Implementation**:
     1. Use HTML forms for filters (e.g., <select> for categories, <input> for price ranges).
     2. Use JavaScript to listen for search input events and dynamically filter the product list.
     3. Display the product list using <div> or <ul> tags, with each item containing an image, name, and price.
3. **Product Detail Page**
   * **Functionality**: Displays detailed information about the selected product, including images, price, and description, with an “Add to Cart” button.
   * **Implementation**:
     1. Use HTML to display product details (e.g., <img> for images, <p> for descriptions).
     2. The “Add to Cart” button should use a <button> tag, with JavaScript sending the product ID and quantity to the backend upon click.
4. **Shopping Cart Page**
   * **Functionality**: Lists all items in the cart with their quantities and total price, allowing users to modify quantities, remove items, or proceed to checkout.
   * **Implementation**:
     1. Use an HTML table (<table>) to display cart items, with each row containing product details, a quantity input field, and a delete button.
     2. Use JavaScript to handle quantity changes and deletions, updating the total price in real time and syncing with the backend.
     3. The “Checkout” button should use a <button> tag to redirect to the checkout page.
5. **Checkout Page**
   * **Functionality**: Collects payment details and shipping information, displays the total cost, and redirects to the order confirmation page upon successful payment.
   * **Implementation**:
     1. Use HTML forms to collect payment and shipping information, with fields such as <input> (text, radio, checkboxes).
     2. Use JavaScript to dynamically calculate and display the total cost.
     3. Redirect to the order confirmation page using JavaScript after successful payment.
6. **Order Status Page**
   * **Functionality**: Provides real-time updates on the shipping status of the order, including a tracking number.
   * **Implementation**:
     1. Use HTML to display order information and shipping status, with the tracking number in a <span> tag.
     2. Use JavaScript to periodically fetch shipping status updates from the backend and refresh the page.
7. **Review/Rating Page**
   * **Functionality**: Allows users to submit feedback and ratings for purchased products and displays reviews from other users.
   * **Implementation**:
     1. Use HTML forms to collect reviews and ratings, with a star-rating component (implemented via a JavaScript library).
     2. Display reviews using <div> or <ul> tags, with each review containing the user’s name, rating, and comment.

**6.2 Accessibility Compliance**

To ensure inclusivity, the user interface adheres to accessibility standards. Below are the implementation details:

* **Responsive Layout**
  + Use CSS media queries (@media) to adapt the layout for different devices.
  + Ensure page elements are properly arranged on all screen sizes, avoiding overflow or overlap.
* **Assistive Technology Support**
  + Add ARIA attributes (e.g., aria-label, role) to interactive elements like buttons and links.
  + Use <button> tags instead of <div> or <span> for buttons to ensure keyboard operability.
* **Color Contrast**
  + Use tools like WebAIM Contrast Checker to verify color contrast compliance with WCAG standards.
  + Ensure text-to-background contrast ratio is at least 4.5:1.
* **Storyboards / Mockups**
  + Use tools like Figma or Sketch to create wireframes that visualize the user journey.
  + For example, illustrate the flow from “Add to Cart” → “Checkout” → “Order Confirmation.”
  + Annotate wireframes with key interactive elements (e.g., buttons, input fields) and page transitions.

**7. Assumptions**

This section outlines all assumptions (what we take for granted) and constraints (hard limits imposed by business, technical, or organizational factors) under which Buyzu is designed and developed.

**7.1 User Base and Traffic Assumptions**

·The system is expected to handle up to 1 million registered users in its initial phase, with peak concurrent users reaching 10,000.

·The system should scale horizontally to accommodate traffic spikes during sales events.

**7.2 Payment and Transaction Processing**

·Payment gateways (Alipay, WeChat Pay, Credit Card) will process transactions within 1-3 seconds under normal conditions.

·Refunds and chargebacks are handled through third-party payment providers rather than internally.

·Orders are finalized only after successful payment confirmation, avoiding inventory discrepancies.

**7.3 Client-Side Compatibility**

·Users must access the platform via modern browsers (Chrome v90+, Firefox v88+, Safari v14+) or mobile devices (iOS 14+/Android 10+). Legacy browsers (e.g., Internet Explorer) are not supported.

·Buyzu assumes users have a basic understanding of online shopping workflows (e.g., adding to cart, checkout).

·The majority of users will access the platform via modern browsers (Chrome, Firefox, Edge, Safari) with JavaScript enabled.

·Mobile users are expected to make up at least 60% of total traffic, requiring a fully responsive UI.

·No sudden DDoS attacks or bot traffic exceeding 5% of total requests. Existing rate-limiting strategies (e.g., token bucket algorithms) ensure service stability.

**7.4 Security and Compliance**

·OAuth 2.0 and JWT are sufficient for authentication and session management.

·PCI-DSS compliance is handled by payment processors, reducing Buyzu's liability in storing sensitive financial data.

·GDPR compliance is assumed, requiring user consent for data processing and offering an account deletion option.

**7.5 Cross-regional compliance assumption**

·There are no major changes in the data sovereignty regulations (such as the Personal Information Protection Act) in the default operating area of the system, and cross-border data transmission (such as the use of AWS Global Accelerator) meets the localized storage requirements of the target market.

**8. Appendices**

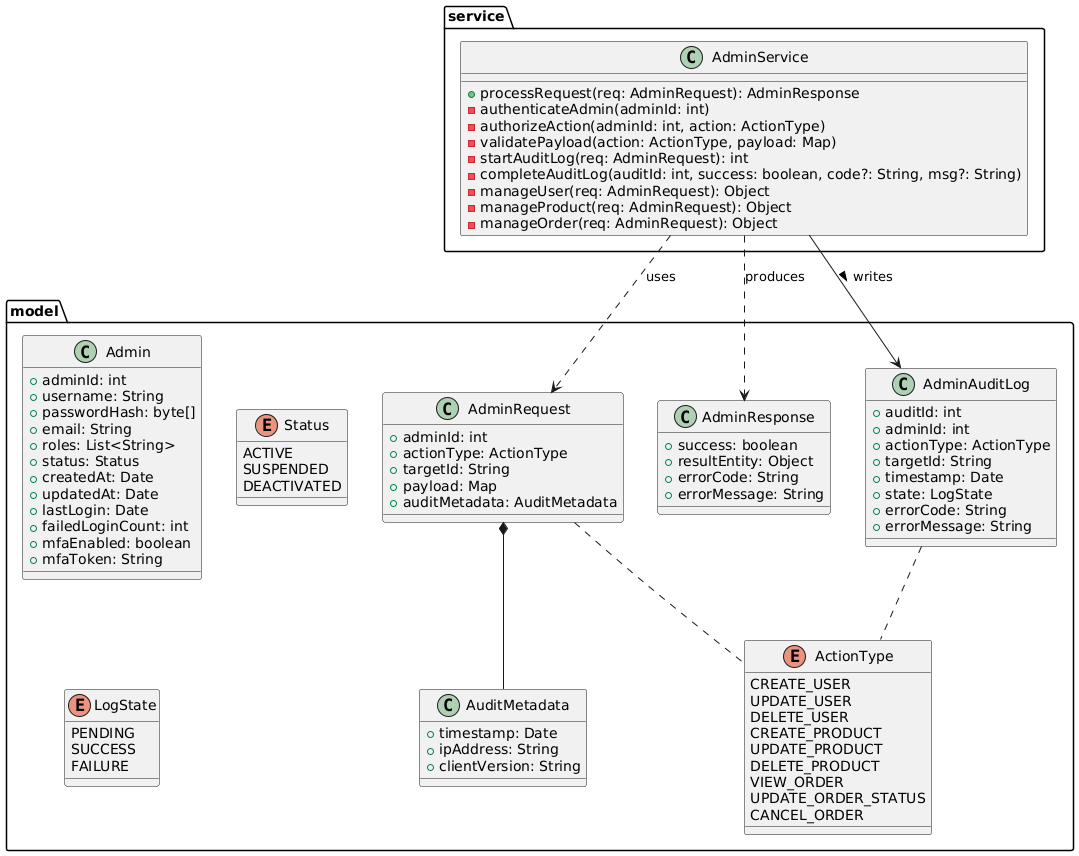
**8.1 UML Class Diagram (Main Components)**

**User Authentication**

一張含有 文字, 收據, 圖表, 平行 的圖片

AI 產生的內容可能不正確。

**Admin [New]**



**Product Recommendation**

一張含有 文字, 圖表, 行, 字型 的圖片

AI 產生的內容可能不正確。

**Shopping Cart**

一張含有 文字, 字型, 圖表, 行 的圖片

AI 產生的內容可能不正確。

**Checkout**

一張含有 文字, 圖表, 行, 字型 的圖片

AI 產生的內容可能不正確。

**8.2 Version History**

| **Version** | **Date** | **Changes** |
| --- | --- | --- |
| 1.0 | 2025-03-11 | Initial Release |
| 2.0 | 2025-05-09 | Final Release |

**9. Conclusion**

This **Design and Implementation Documentation** provides a **condensed yet thorough** overview of the **Buyzu** e-commerce platform. It captures essential architectural decisions, data modeling, interface definitions, component responsibilities, and UI considerations. Further **code-level details** (e.g., exact function names, complex algorithm logic) can be found in the **source repository**’s inline comments and **auto-generated API docs** (e.g., Swagger/OpenAPI).